Generators

Class outline:

- Generators

Generators

Generators

A **generator function** uses `yield` instead of `return`:

```
def evens():
    num = 0
    while num < 10:
        yield num
        num += 2
```

A **generator** is a type of iterator that yields results from a generator function.

Just call the generator function to get back a generator:

```
evengen = evens()

next(evengen)
next(evengen)
next(evengen)
next(evengen)
next(evengen)
next(evengen)
```

Generators

A **generator function** uses `yield` instead of `return`:

```
def evens():
    num = 0
    while num < 10:
        yield num
        num += 2
```

A **generator** is a type of iterator that yields results from a generator function.

Just call the generator function to get back a generator:

```
evengen = evens()

next(evengen)   # 0
next(evengen)   # 2
next(evengen)   # 4
next(evengen)   # 6
next(evengen)   # 8
next(evengen)   # ✕ StopIteration exception
```

How generators work

```python
def evens():
    num = 0
    while num < 2:
        yield num
        num += 2

gen = evens()

next(gen)
next(gen)
```

- When the function is called, Python immediately returns an iterator without entering the function.
- When `next()` is called on the iterator, it executes the body of the generator from the last stopping point up to the next `yield` statement.
- If it finds a `yield` statement, it pauses on the next statement and returns the value of the yielded expression.
- If it doesn't reach a yield statement, it stops at the end of the function and raises a `StopIteration` exception.

View in PythonTutor

Looping over generators

We can use for loops on generators, since generators are just special types of iterators.

```
def evens(start, end):
    num = start + (start % 2)
    while num < end:
        yield num
        num += 2

for num in evens(12, 60):
   print(num)
```

Looping over generators

We can use for loops on generators, since generators are just special types of iterators.

```
def evens(start, end):
    num = start + (start % 2)
    while num < end:
        yield num
        num += 2

for num in evens(12, 60):
   print(num)
```

Looks a lot like...

```
evens = [num for num in range(12, 60) if num % 2 == 0]
# Or  = filter(lambda x: x % 2 == 0, range(12, 60))
for num in evens:
    print(num)
```

Why use generators?

Generators are lazy: they only generate the next item when needed.

Why generate the whole sequence...

```
def find_matches(filename, match):
    matched = []
    for line in open(filename):
        if line.find(match) > -1:
            matched.append(line)
    return matched

matched_lines = find_matches('frankenstein.txt', "!")
matched_lines[0]
matched_lines[1]
```

...if you only want some elements?

```
def find_matches(filename, match):
    for line in open(filename):
        if line.find(match) > -1:
            yield line

line_iter = find_matches('frankenstein.txt', "!")
next(line_iter)
next(line_iter)
```

A large list can cause your program to run out of memory!

Exercise: Countdown

```python
def countdown(n):
    """
    Generate a countdown of numbers from N down to 'blast off!'.
    >>> c = countdown(3)
    >>> next(c)
    3
    >>> next(c)
    2
    >>> next(c)
    1
    >>> next(c)
    'blast off!'
    """
```

Exercise: Countdown (solution)

```
def countdown(n):
    """
    Generate a countdown of numbers from N down to 'blast off!'.
    >>> c = countdown(3)
    >>> next(c)
    3
    >>> next(c)
    2
    >>> next(c)
    1
    >>> next(c)
    'blast off!'
    """
    while n > 0:
        yield n
        n -= 1
    yield "blast off!"
```

Virahanka-Fibonacci generator

Let's transform this function...

```
def virfib(n):
    """Compute the nth Virahanka-Fibonacci number, for N >= 1.
    >>> virfib(6)
    8
    """
    prev = 0  # First Fibonacci number
    curr = 1  # Second Fibonacci number
    k = 1
    while k < n:
        (prev, curr) = (curr, prev + curr)
        k += 1
    return curr
```

..into a generator function!

```
def generate_virfib():
    """Generate the next Virahanka-Fibonacci number.
    >>> g = generate_virfib()
    >>> next(g)
    0
    >>> next(g)
    1
    >>> next(g)
    1
    >>> next(g)
    2
    >>> next(g)
    3
    """
```

Virahanka-Fibonacci generator (solution)

```python
def generate_virfib():
    """Generate the next Virahanka-Fibonacci number.
    >>> g = generate_virfib()
    >>> next(g)
    0
    >>> next(g)
    1
    >>> next(g)
    1
    >>> next(g)
    2
    >>> next(g)
    3
    """
    prev = 0  # First Fibonacci number
    curr = 1  # Second Fibonacci number
    while True:
        yield prev
        (prev, curr) = (curr, prev + curr)
```

Yield from

Yielding from iterables

A `yield from` statement can be used to yield the values from an iterable one at a time.

Instead of...

```
def a_then_b(a, b):
    for item in a:
        yield item
    for item in b:
        yield item

list(a_then_b(["Apples", "Aardvarks"], ["Bananas", "BEARS"]))
```

We can write...

```
def a_then_b(a, b):
    yield from a
    yield from b

list(a_then_b(["Apples", "Aardvarks"], ["Bananas", "BEARS"]))
```

Yielding from generators

A `yield from` can also yield the results of another generator function (which could be itself).

```
def countdown(k):
    if k > 0:
        yield k
        yield from countdown(k - 1)
```

Visualizing countdown

Calls	Executed code	Bindings	Yields
`>>> c = countdown(3)`	`def countdown(k):`	k = 3	

Visualizing countdown

Calls	Executed code	Bindings	Yields
`>>> c = countdown(3)`	`def countdown(k):`	k = 3	
`>>> next(c)`	`  if k > 0:`		
	`    yield k`		3

Visualizing countdown

Calls	Executed code	Bindings	Yields
`>>> c = countdown(3)`	`def countdown(k):`	k = 3	
`>>> next(c)`	`  if k > 0:`		
	`    yield k`		3
`>>> next(c)`	`    yield from countdown(k - 1)`		
	`def countdown(k):`	k = 2	
	`  if k > 0:`		
	`    yield k`		2

Visualizing countdown

Calls	Executed code	Bindings	Yields
`>>> c = countdown(3)`	`def countdown(k):`	k = 3	
`>>> next(c)`	`if k > 0:`		
	`yield k`		3
`>>> next(c)`	`yield from countdown(k - 1)`		
	`def countdown(k):`	k = 2	
	`if k > 0:`		
	`yield k`		2
`>>> next(c)`	`yield from countdown(k - 1)`		
	`def countdown(k):`	k = 1	
	`if k > 0:`		
	`yield k`		1

Visualizing countdown

Calls	Executed code	Bindings	Yields
`>>> c = countdown(3)`	`def countdown(k):`	k = 3	
`>>> next(c)`	`  if k > 0:`		
	`    yield k`		3
`>>> next(c)`	`    yield from countdown(k - 1)`		
	`def countdown(k):`	k = 2	
	`  if k > 0:`		
	`    yield k`		2
`>>> next(c)`	`    yield from countdown(k - 1)`		
	`def countdown(k):`	k = 1	
	`  if k > 0:`		
	`    yield k`		1
`>>> next(c)`	`    yield from countdown(k - 1)`		
	`def countdown(k):`	k = 0	
	`  if k > 0:`		
	`    yield k`		
	`    yield from countdown(k - 1)`		
			StopIteration

Generator functions with returns

Generator function with a return

When a generator function executes a return statement, it exits and cannot yield more values.

```
def f(x):
    yield x
    yield x + 1
    return
    yield x + 3
```

```
list(f(2))
```

Generator function with a return

When a generator function executes a return statement, it exits and cannot yield more values.

```
def f(x):
    yield x
    yield x + 1
    return
    yield x + 3
```

```
list(f(2))   # [2, 3]
```

Generator functions with return values

Python allows you to specify a value to be returned, but this value is not yielded.

```
def g(x):
    yield x
    yield x + 1
    return x + 2
    yield x + 3
```

```
list(g(2))
```

Generator functions with return values

Python allows you to specify a value to be returned, but this value is not yielded.

```
def g(x):
    yield x
    yield x + 1
    return x + 2
    yield x + 3
```

```
list(g(2))  # [2, 3]
```

Generator functions with return values

Python allows you to specify a value to be returned, but this value is not yielded.

```
def g(x):
    yield x
    yield x + 1
    return x + 2
    yield x + 3
```

```
list(g(2))   # [2, 3]
```

It is possible to access that return value, with this one weird trick. But you won't ever need this in 61A!

```
def h(x):
    y = yield from g(x)
    yield y
```

```
list(h(2))
```

Generator functions with return values

Python allows you to specify a value to be returned, but this value is not yielded.

```
def g(x):
    yield x
    yield x + 1
    return x + 2
    yield x + 3
```

```
list(g(2))  # [2, 3]
```

It is possible to access that return value, with this one weird trick. But you won't ever need this in 61A!

```
def h(x):
    y = yield from g(x)
    yield y
```

```
list(h(2))  # [2, 3, 4]
```

Partitions example

(Review) Counting partitions

The number of partitions of a positive integer `n`, using parts up to size `m`, is the number of ways in which `n` can be expressed as the sum of positive integer parts up to `m` in increasing order.

`count_partitions(6, 4)`

2 + 4 = 6
1 + 1 + 4 = 6
3 + 3 = 6
1 + 2 + 3 = 6
1 + 1 + 1 + 3 = 6
2 + 2 + 2 = 6
1 + 1 + 2 + 2 = 6
1 + 1 + 1 + 1 + 2 = 6
1 + 1 + 1 + 1 + 1 + 1 = 6

```
def count_partitions(n, m):
    """
    >>> count_partitions(6, 4)
```

```
    9
    """
    if n < 0 or m == 0:
        return 0
    else:
        exact_match = 0
        if n == m:
            exact_match = 1
        with_m = count_partitions(n-m, m)
        without_m = count_partitions(n, m-1)
        return exact_match + with_m + without_m
```

Converting to a generator

Each call to the generator should yield a partition.

```
def partitions(n, m):
    """List partitions.

    >>> for p in partitions(6, 4): print(p)
    4 + 2
    4 + 1 + 1
    3 + 3
    3 + 2 + 1
    3 + 1 + 1 + 1
    2 + 2 + 2
    2 + 2 + 1 + 1
    2 + 1 + 1 + 1 + 1
    1 + 1 + 1 + 1 + 1 + 1
    """
```

Partitions generator (solution)

```
def partitions(n, m):
    """List partitions.

    >>> for p in partitions(6, 4): print(p)
    4 + 2
    4 + 1 + 1
    3 + 3
    3 + 2 + 1
    3 + 1 + 1 + 1
    2 + 2 + 2
    2 + 2 + 1 + 1
    2 + 1 + 1 + 1 + 1
    1 + 1 + 1 + 1 + 1 + 1
    """
    if n < 0 or m == 0:
        return
    else:
        if n == m:
            yield str(m)
        for p in partitions(n-m, m):
            yield str(m) + " + " + p
        yield from partitions(n, m - 1)
```

Python Project of The Day!

Mathematical Animation Engine

Manim: An open-source Python animation engine for explanatory math videos, first created by Grant Sanderson for 3Blue1Brown videos.

Check out the examples gallery, oooo!